Newcastle University e-prints

Lessons from history?

I have long had a strong interest in the history of computing, though my historical researches have concentrated on the origins of computers, rather than their more recent development. In recent years I have had little time to pursue this interest, but two events have driven me to think more carefully about the more recent past, and to wonder what lessons can be drawn from it.

The first of these events was an invitation, which I was too flattered to turn down, to rewrite the final section of that splendid lavishly-illustrated history, "A Computer Perspective", which had been out of print since soon after it was originally published in 1973. My task was to replace a four page account of the period 1950-1970 with an account of the period 1950-1990 - but still in just four pages. The second event was an invitation from the IEE to give a lecture at their planned Charles Babbage Bi-Centenary Conference, summarizing the entire development of programming from Babbage to modern times. This also I accepted, though I should have turned it down, if only because the conference was cancelled at the last moment.

The result, however, was that I was forced to think very hard about the history of the last forty years or so, and about what if anything we can learn from it. It was a sobering experience when I reluctantly reached the decision that, despite all the software research and development that has been undertaken, the two most important events in the history of software since the invention of the stored program concept were the development of (i) FORTRAN, and (ii) the microcomputer! (The former established the recognition that it was practicable to program at a level which was significantly more abstract than the machine code level; the latter caused what was in effect a huge programmer and user population explosion. This led to such a great increase in the number and variety of creative new applications, and also to so much production and marketing of lots of essentially similar software systems that the normal processes of competition and evolution at last started to work fairly effectively.)

Between these two events, I regard the late 1960s as somewhat of a watershed. Not only was this when IBM unbundled its software, so creating the first real possibility of a software market and indeed an industry. This was when two NATO Conferences identified what was termed "the software crisis" and first promoted the concept of, and the need for, a body of knowledge and practice worthy of the term "software engineering". It was also the time when the concepts of software components and software re-use began to be talked about seriously, and when the technology that is currently touted as central to these concepts, namely "object-oriented programming" had its real origins, in SIMULA-67. However to my mind the first widely successful use of software componentry occurred shortly afterwards. This was the application of a much simpler scheme, namely the pipes and filters of UNIX, to the valuable but special case of applications which could be cast in terms of operations on linear character streams.

The software design task centres on the problem of mastering complexity - the new complexity that each new application provides. The classic (perhaps the only) technique for dealing with complexity is of course "divide and conquer" Some developments have, so to speak, divided off part of the complexity, and provided a canned solution of general utility (e.g. to the problems of dealing with physical storage, or the fact that the application is being carried out on a set of computers rather than a single one). This sort of development can often be employed quite readily, and so will be taken up relatively quickly. But only gradually have various commonly useful standard facilities been identified, and provided in this way.

Other developments are more related to the actual division process. For example UNIX pipes and filters, or object-oriented programming for that matter, really just provide a convenient form of mortar, with which the programmer can glue together whatever bricks he or she chooses to invent or obtain. But a new form of mortar does not help one to invent an improved set of bricks - the subdivision task (equivalently, the task of *choosing* interfaces, or of *inventing* languages) remains and still requires creativity (or at least an ability to recognize that some pre-existing component would suffice). Neither for that matter does a new, even completely formal, method of specifying such interfaces or languages directly aid the task of choosing just what is

to be specified. Thus creativity of course continues to play a pivotal role in the software development process.

Creativity is difficult to inculcate, to improve significantly, or to schedule (leave alone to automate) and varies greatly between individuals. This is as true now as it was twenty-five years ago. Thus it is perhaps not surprising that, to reiterate my earlier point, it is my regretful conclusion that the last twenty-five years of research and development into software engineering, programming methodology and the like, though they have produced much of merit, have not had as significant an impact as an event which is quite external to them, namely the advent of the microcomputer, which caused such an increase in the number of people involved in the creative process.

It is always dangerous to use the lessons of the past in an attempt to predict the future - but they are probably the best basis available. I am reasonably sure that software engineering, and in particular software componentry and reuse, will develop and be aided by the various efforts now going into object-oriented design tools, etc. But progress in establishing effective and widely-usable object libraries will I fear be much slower than most people hope and expect - because of the creativity required. Indeed my strong inclination is to predict that the next ten years, say, will again see the software world affected more by other external events (some of which one can already see starting to happen, or looming on the horizon), or by some unexpected new idea, than by its general research and development into software engineering and programming methodology.

The past as millstone

The past is not only a source, albeit of dubious trustworthiness, of guidance as to the likely future, it is also - especially in the case of software - an increasing source of impediments to change, if not to progress. (Such impediments have in recent years being engagingly termed "legacy systems".) Most serious software cannot be designed for a "green field site". It has to support old interfaces, old data formats, old methods of working, etc., as well as provide some improvements. Only by such means can it fit in alongside existing investments in hardware, data, software and people, and produce an overall economic benefit.

The speed with which new developments will be taken up will thus often depend in large part on the effectiveness with which they have been designed to cope with the past. A striking example of success in this regard is World Wide Web - one of the most elegant aspects of its design, contributing greatly to its rapid spread, is the way it enables continued use of a variety of existing systems such as Gopher, FTP, WAIS, etc. And the operating system battles now raging will evidently be won in part by demonstrating successful support of existing applications.

Living with the past is even more complicated when the problem is to introduce new software into a continuously running system. This problem has been faced most severely by the telephone companies - their level of success is remarkable, but by no means total, as recent major system failures have shown.

Predicting the effect of these problems on the various segments of the software industry is not easy. Logically, one might argue that legacy systems are by definition an ever-growing impediment, and hence will slow down development more and more. But who knows what novel techniques for coping with such systems might be developed, and prove so effective that the millstone proves ineffective. The very invisibility of software is a great aid here. One can glue ill-fitting systems and components together by a byzantine network of emulators, converters, filters, etc., and rely on computer power to reduce their performance impact to a level which is acceptable. (The fact that the equivalent design represented in visible hardware components would probably arouse ridicule is perhaps best ignored!)

The characteristics of software

The most fundamental characteristics of software over and above the fact that it typically embodies immense complexity, are (i) that it is intensely brittle, and (ii) once developed, it can be replicated at virtually zero cost. The brittleness is of course logical in nature, and comes from its digital character. No recourse can be made to normal notions of continuity and of safety factors in designing the software, in validating it, in estimating its dependability, etc. Research on probabilistic algorithms, and on inherently self-checking programs, could be regarded as embodying a slight move away from this situation, but logical brittleness is likely to remain a major characteristic of software, and continue to impact our ways of producing and using it. I foresee at best only slow incremental improvement in our ability to deal with this characteristic - and find it difficult to imagine developments external to the software world which will have much impact on this situation.

The extremely low replication cost comes from the technologies, all digital in nature, that we have for embodying software - technologies that are being increasingly used for many other sorts of information. The use of these technologies provides a set of commercial and legal problems (termed the problems of "digitized property" by John Perry Barlow) to all involved in any form of information storage and transfer, from libraries to film producers, from book publishers to software houses.

Developments in networking, in security mechanisms, in copyright law, in patenting practices, etc., could be equally relevant to all forms of digitally-encoded information, and could have a major impact - not least on the various components of the software industry. What these developments are likely to be is something I leave to people more expert in the various fields to predict - I content myself by predicting that one or more of these developments is likely to be the major determinant of the future of the software industry.

Of course, aside from these general characteristics, one can identify a number of characteristics which effectively segment the software field. For example, the problems of producing safety-critical software for an individual process control application are very different from those of producing software for embedding in some mass-produced household appliance, which are in turn very different from those of successfully competing in the shrink-wrap software market. The relative seriousness of the various Issues that I have discussed, and hence of predicting their impact on future developments, will vary across these different segments. But discussion of this point goes beyond the intended scope of this brief position paper.

Concluding remarks

By way of concluding remarks, let me simply use the following quotation:

> The future is dark, the present burdensome. Only the past, dead and finished, bears contemplation. Those who look upon it have survived it; they are its product and its victors. No wonder therefore that men concern themselves with history.
>
> G.R. Elton. *The Practice of History* (1967)

This quotation is one that I have used once before. This was in 1978 in an Invited Paper, at the International Conference on Software Engineering, in which I surveyed developments since the NATO Conference ten years earlier which had first launched the Software Engineering bandwagon. In fact, though I very much like this quotation, and felt that it had a similar potential relevance to our workshop and so could not forbear from including it here, I actually did not and do not subscribe to the sentiments it expresses. I study history because I find it enjoyable, not as an escape from the future - which in fact I find even more fascinating to contemplate.